Programming Techniques and Data Structures

M. Douglas McIlroy*
Editor

Distributed Computation on Graphs: Shortest Path Algorithms

K. M. Chandy and J. Misra
University of Texas at Austin

We use the paradigm of diffusing computation, introduced by Dijkstra and Scholten, to solve a class of graph problems. We present a detailed solution to the problem of computing shortest paths from a single vertex to all other vertices, in the presence of negative cycles.

CR Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems; D.1.3 **[Programming Techniques]**: Concurrent Programming; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems; G.2.2 **[Discrete Mathematics]**: Graph Theory

General Terms: Algorithm, Theory

Additional Key Words and Phrases: distributed computation, shortest path, negative cycle, depth first search, diffusing computation

1. Introduction

This paper presents distributed algorithms based on the work of Dijkstra and Scholten [1], for solving graph problems using networks of communicating processes. The solution to one particular graph problem, that of finding shortest paths from a single vertex to all other vertices in a weighted, directed graph, in the presence of negative cycles, is discussed in detail. We then show how this solution may be applied to other graph problems including depth-first search in an undirected graph.

Our model of computation is a network of processes in which processes communicate only by sending and receiving messages; the model is presented in detail in Sec. 2. We describe the classical shortest path problem [2] and the necessary terminology from graph theory in Sec. 3. The distributed algorithm is given in Sec. 4 and its proof in Sec. 5. Applications to other graph problems are discussed in Sec. 6.

2. Model of a Network of Communicating Processes

A process is a sequential program which can communicate with other processes by sending/receiving messages. Two processes P and Q are said to be *neighbors* if they can communicate directly with one another without having messages go through intermediate processes. We assume that communication channels are bidirectional: if P can send messages to Q then Q can send messages to P. A process knows the identities of its neighbors; otherwise it is ignorant of the identities of all other processes and of the general structure of the network.

We assume a very simple protocol for message communication; this protocol is equivalent to the one used by Dijkstra and Scholten [1]. Every process has an input buffer of *unbounded* length. If process P sends a message to a neighbor process Q, then the message gets appended at the end of the input buffer of Q after a finite, arbitrary delay. We assume that (1) messages are not lost or altered during transmission, (2) messages sent from P to Q arrive at Q's input buffer in the order sent, and (3) two messages arriving simultaneously at an input buffer are ordered arbitrarily and appended to the buffer. A process receives a message by removing one from its input buffer.

The assumption of unbounded length buffers is for ease of exposition. We show, in Sec. 6, that for our problem the input buffer length of process Q can be bounded by the number of neighbors of Q.

3. The Shortest Path Problem

$G = (V, E)$ is a directed graph in which V is the set of vertices and E is the set of edges. Edge (v_i, v_j) has an associated length w_{ij}. If edge (v_i, v_j) exists then v_j is said to be a *successor* of v_i and v_i is said to be a *predecessor* of v_j. It is required to determine lengths of the shortest paths from a special vertex v_1 in V to all other vertices in V.[1] Since some w_{ij} may be negative, a cycle of negative total length (called a *negative cycle*) may exist in the graph. If a negative cycle is reachable from v_1 then all vertices reachable from that negative cycle will have a shortest path length of $-\infty$. The *distance* of a vertex v_i is the length of the shortest path from v_1 to v_i and is denoted by L_i.

* Former editor of Programming Techniques and Data Structures, of which Ellis Horowitz is the current editor.

This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR 81-0205.

Authors' Present Address: K. Mani Chandy and J. Misra, Computer Sciences Department, University of Texas, Austin, TX 78712.

[1] We assume familiarity with graph theoretic terms such as path, shortest path, etc.

4. A Distributed Algorithm for the Shortest Path Problem

Consider a network of processes corresponding to graph G; process p_i represents vertex v_i, for all i, and p_i and p_j are neighbors if edges (v_i, v_j) or (v_j, v_i) exist in G. p_i knows the weight w_{ij} for every outgoing edge (v_i, v_j). However, p_i may not know the weights of incoming edges or the identities of processes other than its neighbors.

Process p_1 initiates a computation to determine the lengths of shortest paths from v_1 to all vertices. In the following, we use vertex v_i and process p_i interchangeably when no confusion can result.

4.1 The Structure of the Algorithm

The algorithm works in two phases, both of which are initiated by p_1. At the end of phase I, every process p_i will have the value of L_i, if $L_i \neq -\infty$. If for some vertex v_j, $j \neq 1$, $L_j = -\infty$ then p_j will *not* be aware of this fact at the end of phase I; the goal of phase II is to inform all such processes that they are at distances of $-\infty$.

4.2. The Structure of Phase I Computation

4.2.1 Messages Used in Phase I

Phase I computation uses two kinds of messages:

(1) A *length message* is a two-tuple (s, p), where p is the identity of the process sending the message and s is a number. p_i sends a length message (s, p_i) to p_j to inform p_j that there is a path of length s from v_1 to v_j in which v_i is the prefinal vertex.

(2) An *acknowledgment message* or *ack* has no other data associated with it. A process p_j sends an ack to a process p_i in response to a length message sent by p_i. Intuitively, an ack denotes that the length sent by p_i to p_j has been (or will be) taken into consideration by all processes reachable from p_j.

A process p_i, $i \neq 1$, maintains a local variable d which denotes the length of the shortest path received so far by p_i. Upon receiving a length s from a predecessor, if $s < d$, p_i sets d to s and in this case it sends a length message $(s + w_{ij}, p_i)$ to every successor p_j. It may seem that acks are superfluous. Clearly length messages can be used to compute successively shorter paths. However, the presence of negative cycles means that this will be a nonterminating computation. Acks are used to terminate phase I computation as described below.

4.2.2 Local Data Used by a Process p_i During Phase I

Each process p_i uses three local variables:

d This is the shortest length of paths from v_1 to v_i *known to this process at this point in the computation*; $d = \infty$ if no length message has been received.

pred This is the predecessor from which the length d was received; this is the prefinal vertex on the shortest path to v_i computed so far. *pred* is undefined if $d = \infty$ or $i = 1$.

num This is the number of unacknowledged messages, that is, the number of messages sent by this process for which no ack has been received so far.

4.2.3 Phase I Algorithm for Process p_j, $j \neq 1$

```
Initialization
    {no length message has been received; there are no unacknowledged
     messages}
begin $d := \infty$; pred is undefined; num := 0 end;

Upon receiving a length message $(s, p_i)$
if $s < d$ then
begin
    {send an ack to pred, the prefinal vertex on the previous shortest
     path, if it has not been sent already}
  if num > 0 then send an ack to pred;
    {update d, pred}
  pred := $p_i$; $d := s$;
    {send length messages to all successors of $v_j$ and increment num
     appropriately and then return ack to pred if num = 0}
  send a length message $(d + w_{jk}, p_j)$ to every successor $p_k$;
  num := num + the number of successors of $v_j$;
  if num = 0 then send an ack to pred
end
else {$s \geq d$} {new length does not denote a shorter path}
  send ack to $p_i$.

Upon receiving an ack from process $p_k$
begin
    {decrement number of unacknowledged messages}
  num := num − 1;
    {send acknowledgement to pred if acks have been received for all
     messages}
  if num = 0 then send ack to pred
end.
```

Note. If *num* > 0 at any time, then a process has exactly one message to which it has not sent an ack, and this ack should go to *pred*.

4.2.4 Initiation of Phase I

4.2.4.1 Phase I algorithm for process p_1

```
Initialization
  $d := 0$; pred is undefined;
  send $(w_{1k}, p_1)$ to all successors $p_k$; num := number of successors of $v_1$.

Upon receipt of a length message $(s, p_i)$
  {start phase II if a negative cycle is detected}
if $s < 0$ then terminate phase I and start phase II
        else  return ack to $p_i$
Upon receiving an ack
  {update num; start phase II if there is no unacknowledged message
   remaining}
num := num − 1;
if num = 0 then terminate phase I and start phase II.
```

4.2.5 Example

Consider the graph shown in Figure 1. Four feasible snapshots of the network showing possible values for *d*, *pred*, and *num* for the six processes in this example are shown below. Since transmission delays are arbitrary, network computation is nondeterministic. Hence the four snapshots shown below form only one of many sequences which may arise. The question mark denotes an undefined value for *pred*.

Snapshot 1. p_1 has sent one message to each of p_2 and p_3 which have not yet been received.

	1	2	3	4	5	6
d	0	∞	∞	∞	∞	∞
pred	?	?	?	?	?	?
num	2	0	0	0	0	0

Snapshot 2. p_2, p_3 have received length messages (3, p_1), (4, p_1), respectively. p_3 has sent (10, p_3) to p_4, which p_4 has received.

	1	2	3	4	5	6
d	0	3	4	10	∞	∞
pred	?	1	1	3	?	?
num	2	0	1	0	0	0

Snapshot 3. p_5, p_6 receive (11, p_4), (12, p_4), respectively, from p_4. p_6 sends an ack to p_4; this ack is received by p_4. p_4 receives (5, p_2). Next p_4 sends an ack to p_3, which is received, and sends (6, p_4), (7, p_4) to p_5 and p_6, respectively, which they both receive. p_5 sends an ack to p_4 which is received by p_4.

	1	2	3	4	5	6
d	0	3	4	5	6	7
pred	?	1	1	2	4	4
num	2	1	0	2	0	0

Snapshot 4. p_3 sends an ack to p_1 since p_3's *num* is zero. p_5 sends (2, p_5) to p_2, thus causing p_2 to send an ack to p_1. The acks are received since p_1 has no further unacknowledged messages it terminates phase I.

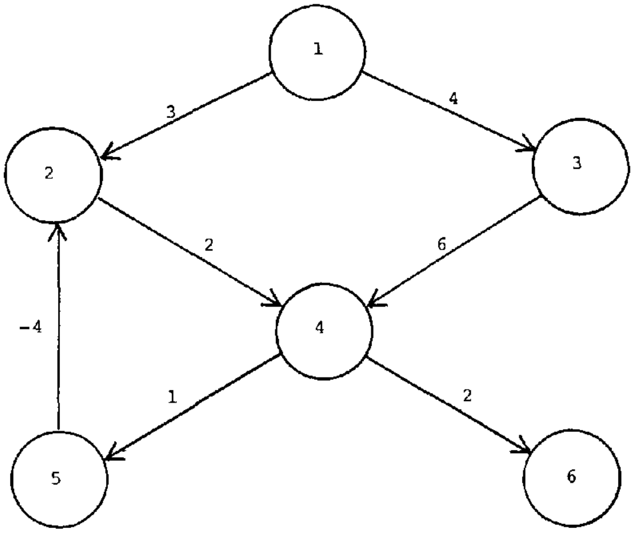
	1	2	3	4	5	6
d	0	2	4	5	6	7
pred	?	5	1	2	4	4
num	0	1	0	2	1	0

4.3 The Structure of Phase II Computation

4.3.1 Messages Used in Phase II

Phase II employs two kinds of messages: over? and over-. An over- message is sent by process *j* to all its successors if process *j* has determined that phase I is over and $L_j = -\infty$; an over- message orders the recipient to halt all phase I computation (if it has not done so already), set its *d* to $-\infty$ and propagate the over- message to its successors. If a process already has its $d = -\infty$ when it receives an over- message, it takes no action. An over? message is sent by process *j* to all its successors when it has determined that phase I is over, but has not determined whether $L_j = -\infty$. An over? message orders the recipient to halt all phase I computation. If the recipient p_i has *num* = 0 it sends over? messages to its successors; otherwise (if p_i has *num* > 0) it can be shown that $L_i = -\infty$, and therefore p_i sets its $d := -\infty$ and sends over- to its successors. Note that it is redundant for any process p_i to send duplicate messages to a process p_j or to send over? after over-. Every process other than p_1 will receive an over? or an over- message.

Fig. 1. A Network with Weighted Edges.



4.3.2 Detailed Algorithm for Phase II

4.3.2.1 Initiation of Phase II by Process p_1

```
if $p_1$ receives a message $(s, p)$, with $s < 0$, during phase I
  then {$p_1$ detects that it is in a negative cycle}
    send an over- message to all its successors
  else {$num = 0$ for $p_1$ at the end of phase I}
    send over? message to all successors.
```

4.3.2.2 Phase II Algorithm for Process p_j, $j \neq 1$ with $num_j > 0$

```
Upon receiving a phase II message (over- or over?)
if $d \neq -\infty$ then
  begin $d := -\infty$;
      send over- to all successors
  end.
```

4.3.2.3 Phase II Algorithm for Process p_j, $j \neq 1$ with $num_j = 0$

```
Upon receiving an over- message
if $d \neq -\infty$ then
  begin $d := -\infty$;
      send over- to all successors
end.
```

```
Upon receiving an over? message

if $d \neq -\infty$ then send over? to all successors who have not been sent
such a message.
```

5. Proof of Correctness

We define v_i to be a *finite* vertex if $L_i \neq -\infty$; v_i is an *infinite* vertex if $L_i = -\infty$.

LEMMA 1. *For any j, $L_j \leq d_j$ at all times.*

PROOF. We observe that every d_j is the length of some path from v_1 to v_j.

LEMMA 2. *If there is a finite path of length d_j^* to a vertex v_j, then from some point onward in the computation $d_j \leq d_j^*$, if phase I does not terminate.*

PROOF. Proof is by induction on the number of edges on the path. Lemma 2 is trivial when the number of edges in the path is zero. Now assume Lemma 2 holds for all paths with k or fewer edges. Consider a path with $k + 1$ edges from v_1 to v_j in which v_i is the prefinal vertex and the path length to v_i is $d_i^* = d_j^* - w_{ij}$. From the induction hypothesis eventually, $d_i \leq d_i^* = d_j^* - w_{ij}$; therefore p_j will eventually receive $(d_i + w_{ij}, p_i)$ which guarantees that $d_j \leq d_i + w_{ij} \leq d_j^*$. It follows from the algorithm that d_j can never increase. Therefore, $d_j \leq d_j^*$ from that point onward in the computation.

LEMMA 3. *If phase I does not terminate then from some point onward in the computation, every infinite vertex v_j will have an infinite vertex for $pred_j$ and every finite vertex v_j will have a finite vertex for $pred_j$, $j \neq 1$.*

PROOF. The following holds for all j, $j \neq 1$, at all times:

$$d_i + w_{ij} \leq d_j \qquad \text{if } i = pred_j.$$

From Lemma 1, $L_i \leq d_i$, for all i. Therefore,

$$L_i + w_{ij} \leq d_j, \qquad \text{if } i = pred_j.$$

If v_j is infinite then from Lemma 2, eventually d_j gets arbitrarily small. In particular, from some point onward in the computation, for every finite v_i,

$$d_j < L_i + w_{ij}.$$

Hence from that point onward $pred_j$ will be an infinite vertex.

From Lemmas 1 and 2, if phase I does not terminate then eventually every finite v_i will have $d_i = L_i$ and $pred_i$ will be the prefinal vertex on this path; $pred_i$ must therefore be a finite vertex.

THEOREM 1. *Phase I terminates.*

PROOF. Assume phase I never terminates. Then $d_j = L_j$ for every finite vertex v_j from some point in phase I computation and hence no finite vertex sends a length message from then on. From Lemma 3, finite vertices eventually form a rooted directed tree where $pred_j$ is the father of v_j, $j \neq 1$, and v_1 is the root. A leaf vertex v_j, $j \neq 1$, in this tree cannot be the *pred* for any finite vertex (since it is a tree) nor can it be the *pred* for any infinite vertex, from Lemma 3; therefore eventually $num_j = 0$ and v_j will send an ack to $pred_j$. Induct on the height of the tree to show that every finite vertex will eventually have $num = 0$. If p_1 is a finite vertex it will then terminate phase I computation. If p_1 is an infinite vertex, from Lemma 2, it will eventually detect that it is in a negative cycle and hence terminate phase I. Hence phase I will terminate! Contradiction!

THEOREM 2. *At the termination of phase I,*

1. *if v_j is a finite vertex, $d_j = L_j$ and $num_j = 0$;*
2. *if v_j is an infinite vertex, then and only then, there is some v_i such that there is a path from v_1 to v_j through v_i, in the graph, and $num_i > 0$.*

PROOF. (1) For a finite vertex v_j, we define $e(j)$ to be the number of edges on a shortest path from v_1 to v_j (if there are several shortest paths we choose the shortest loop-free path with maximum number of edges). The result follows by induction on all vertices v_j with $e(j) \leq k$, for $k = 0, 1, 2. \ldots$

(2) Assume the contrary that for an infinite vertex v_j, every vertex v_i on a path from v_1 to v_j has $num_i = 0$, at the end of phase I. Even if p_1 did not terminate phase I computation, v_j will never receive a length message and thus d_j will not decrease. This contradicts Lemma 2. The other part of the proof follows by similar arguments.

THEOREM 3. *Phase II terminates and at that point $d_j = L_j$ for every vertex v_j.*

PROOF. Phase II terminates since any process sends at most 2 messages: over? followed by an over- message. No finite vertex receives an over- message because there cannot be an infinite vertex on a path from v_1 to a finite vertex. Therefore d_j remains unchanged during phase II for a finite vertex; and from Theorem 2, $d_j = L_j$ at the beginning of phase II. For an infinite vertex v_j, there is a path from v_1 to v_j through v_i, where $num_i > 0$ at the end of phase I. Therefore p_i will propagate an over-message once it receives any phase II message, and therefore $d_j = -\infty = L_j$ eventually.

6. Notes on the Algorithm

6.1 Unbounded Buffers

A process p_i sends (strictly) monotone decreasing lengths in every length message to any other process p_j. Therefore any length message sent by p_i can overwrite any earlier message sent by p_i which is still in the buffer. Hence p_j need only store one message (the latest message) from each predecessor. The space requirement for acks can be reduced by storing the *number* of acks sent from p_j to p_i, which are still in the buffer; this number is incremented by 1 each time p_j sends an ack to p_i. p_i can remove multiple acks from the buffer and reduce num_i accordingly. Hence we need space for at most one message and one ack count for every neighbor of a process p_j in the input buffer of p_j.

6.2 Applications to Other Graph Problems

A number of other graph problems can be formulated as shortest path problems using a more general notion of path length. We define a path length function $\not\ell$, a real valued function on paths, starting from v_1, as follows:

$$f[\text{path with no edges}] = 0$$

$$f[P_i; (i, j)] = g_i(f(P_i), w_{ij}),$$

where P_i is any path from v_1 to v_i, P_i; (i, j) is the path P_i followed by edge (v_i, v_j), g_i is any arbitrary computable function which is monotone in the first argument, and w_{ij} is some given real number denoting the "length" of edge (v_1, v_j).

The shortest path algorithm of Sec. 4 can be used to compute

$$d_j = \min\{f(P_j) \mid P_j \text{ is a path from } v_1 \text{ to } v_j\}, \qquad \text{for all } j.$$

The only change is in phase I computation in the content of the length message sent; instead of p_j sending $(d_j + w_{jk}, p_j)$ to a successor p_k, it now sends $(g_j(d_j, w_{jk}), p_j))$. Monotonicity of g in the first argument is essential, since it guarantees that every process sends monotone decreasing path lengths, if it receives monotone decreasing path lengths.

We list some graph problems and show how they can be solved under this shortest path formulation.

(1) Find all vertices reachable from v_i. We wish to set d_j to 0 if v_j is reachable from v_1; else set d_j to ∞. We use the following function,

$$g_i(x, y) = x.$$

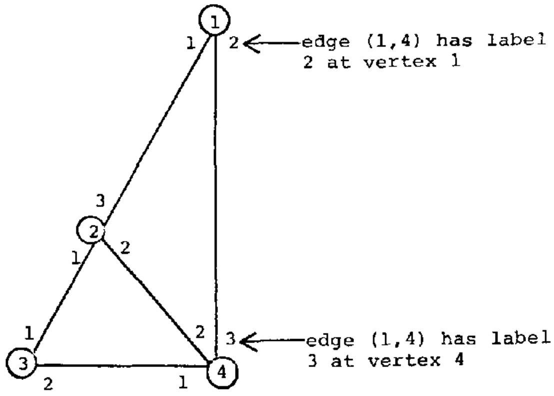
(2) Find all vertices which can reach v_1. This is the same as (1), except length messages are sent to predecessors.

(3) Find the maximum strongly connected component. Determine if a given vertex v_1 is in a nontrivial strongly connected component: use both (1) and (2). A separate computation is then needed to determine whether there is a vertex which has its d set to 0 in both computations.

(4) Construct a depth-first search tree. Consider an undirected graph G. For each vertex j label all the edges incident on j with 1, 2, 3, ... In a depth-first search we would normally label the "left-most" edge on j with 1, the next left-most edge 2, and so on. (However, for purposes of proof the labeling is arbitrary.) Note that edge (i, j) may be the rth left-most edge incident on i and the sth left-most edge incident on j and it is not necessary that $r = s$. An example is shown below.

In a depth-first search starting from a vertex (say vertex 1), the vertices of the graph are traversed beginning with a depth-first search of the left-most successor of vertex 1. The collection of paths traversed to reach each vertex for the *first* time forms a tree called the depth-first search tree. In the above example the depth-first search tree has edges (1, 2), (2, 3), and (3, 4). Our goal is to determine the depth-first search tree; in particular we want to determine the path leading to every vertex in the depth-first search tree.

Fig. 2. An Undirected Graph with Labeled Edges: An Application of Depth-First Search.



edge (1,4) has label 2 at vertex 1

edge (1,4) has label 3 at vertex 4

Let P be a path $(i_1, \ldots, i_k)$. Then define $f(P) = (j_1 \ldots, j_{k-1})$, where j_m, $m = 1, \ldots, k - 1$, is the label assigned to edge (i_m, i_{m+1}) at vertex i_m. In our example, if $P = (1, 2, 3, 4)$ then $f(P) = (1, 1, 2)$.

Let $f(P) = (j_1, \ldots, j_m)$ and $f(P') = (k_1, \ldots, k_n)$. We define $f(P) < f(P')$ if and only if either

(i) for some r, $j_r < k_r$ and $j_i = k_i$ for $i = 1, \ldots, r - 1$, or

(ii) $n > m$ and $j_i = k_i$ for $i = 1, \ldots, m$.

Thus (1, 2, 3) < (3) and (1) < (1, 1, 2, 2).

It is evident that $d_j = \min\{f(P_j) \mid P_j$ is a path from v_1 to $v_j\}$ denotes the path in the depth-first search tree up to v_j.

6.3 Earlier Work

The algorithm suggested in this paper is a modification of an algorithm proposed by Dijkstra and Scholten [1] for termination detection of a class of distributed computations, called diffusing computations. In their algorithm $pred_j$ does not change as long as $num_j > 0$; the algorithm terminates when $num_j = 0$ for every p_j. We allow $pred_j$ to change while $num_j > 0$; this allows us to terminate the phase I algorithm even when some $num_j > 0$. This is critical for identifying infinite vertices since those are the ones which are reachable from a vertex with $num > 0$.

Acknowledgments. We are indebted to E. W. Dijkstra for his comments on an earlier draft of this paper; his suggestions led to more concise proofs in Section 5. We are also grateful to unknown referees and M. D. McIlroy for their suggestions and corrections.

Received 7/80; revised 9/81; accepted 3/82

References

1. Dijkstra, E.W., and Scholten, C.S. Termination detection for diffusing computations. *Inf. Process. Lett 11,* (Aug. 1980), 1, 1–4.
2. Ford, L.R., and Fulkerson, D.R. *Flows in Networks.* Princeton Univ. Press, Princeton, N. J., 1962.